METHOD AND APPARATUS FOR INCREASING TRANSACTION CONCURRENCY BY EARLY RELEASE OF LOCKS IN GROUPS

FIELD OF INVENTION

The invention relates to the field of data processing systems. More specifically, the invention relates to locking techniques used for database management systems. More particularly, the invention relates to controlling the locks acquired on various data records and release of locks in groups.

BACKGROUND

A database is a collection of related data items. To perform various tasks, users access data items stored in the database via transactions. These user accesses need to be controlled to ensure data consistency.

A database management system (DBMS) supports transactions to access the data items stored in the database. To get a consistent view of the data items, the transaction acquires locks on the data items before accessing them. The transaction may release these locks when it no longer needs access permissions to the data item. By default, all locks that the transaction owns are released when the transaction completes. Thus it can be inferred that every lock that the transaction acquires has an associated life-span. This life-span is the time elapsed between when the lock is acquired and when the lock is released. This lock life-span is commonly referred to as a lock duration. The lock duration is typically shorter than the life span of the lock acquiring transaction. A lock manager (LM) processes locking and unlocking requests by transactions. The LM should also support the ability to release locks in a given group before the transaction finishes.

Transactions have an additional property called isolation. Generally, DBMSs support serializable, cursor stable, read committed, dirty read, etc., isolation levels. For most non serializable isolation levels, locks may be released on data items before the transaction finishes. One such example is a read committed isolation level. According to the consistency semantics of this isolation level, the most recently committed value of the given data item should be read by the accessing transaction. One implication of this is that the transaction may request a lock on a data item, read the data item, and then release the lock on the data item, and this may be repeated as many times as desired, where each read of the same data item may potentially return a different value. Such a transaction may need to group locks and release the group locks in one call to the LM, instead of releasing all such locks individually.

Commercially available DBMSs support a few lock durations. Each duration has a distinct name and predefined semantics. For example, instant, short, medium and long lock durations are used in many commercially available DBMSs. The duration of a lock is not determined by the LM, but by the transaction that requests the lock. A transaction may, based on its isolation level, choose different durations for the same data item. If the transaction is executing under serializable isolation, then the lock durations are trivially from the time acquired to the end of the transaction. In this case, all locks for a transaction are released when the transaction execution is complete.

However, for weaker isolations (cursor stability, read-committed), lock durations play a very important role. Transactions may request release for some locks before transactions are complete. Such early release of locks is necessary to promote transaction concurrency. Reducing lock duration reduces access conflicts. For example, a transaction may request locks that are to be released early in the medium duration. When the transaction determines

that access is no longer needed to any data item in that set, it may request the LM to release all medium duration locks.

However, there is currently no meaningful way to define and manage a potentially unlimited number of groups which a transaction may want to classify a given lock. Managing these groups with a few a priori meaningful lock durations will necessitate a separate queue manager that will need to note all classifications. This would be cumbersome and complicated. Providing only a few lock durations may not be suitable to effectively manipulate lock usage for complex non serializable transactions.

The present invention addresses this and other problems associated with the prior art.

SUMMARY OF THE INVENTION

Activity Duration Locking (ADL) provides an efficient simple solution to manage a potentially large number of concurrently active lock groups. The ADL scheme allows a transaction fine control over creation of lock groups and classification of locks into lock groups enabling faster releasing of locks in groups. The life time that the lock group is active defines a new lock duration. This allows a large number of lock durations to be defined with the creation of each new lock group, and any number of lock durations can be active for a given transaction. A very general concept of lock duration is supported without using any a priori semantics to individual lock durations and the duration of a lock group may be determined by the transaction as desired.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. 1 is a block diagram of a Database Management System that uses an Activity Duration Locking (ADL) scheme.

FIG. 2 is a logic diagram showing a duration of multiple activities for a transaction.

FIG. 3 is a block diagram showing how the multiple activities are identified.

FIG. 4 is a block diagram showing how a group of tuples are associated with an activity in a prefetch operation.

FIG. 5 is a block diagram showing how locks are assigned to different activities.

FIG. 6 shows how activity identifiers are assigned.

FIG. 7 is a flow diagram showing how locks are released for different activities.

DETAILED DESCRIPTION

FIG. 1 shows a database system that includes a database 8 and a processor 4 (Central Processing Unit (CPU) and memory) that stores and executes software for a Database Management System (DBMS) 6. A computing device 2 containing processor 4 in one example is a server. The processor 4 may be a single CPU or multiple CPUs that operate in the server 2. The database 8 is typically stored on a plurality of Direct Access Storage Devices (DASDs) although only one such device is illustrated in FIG. 1. The DASDs may be, for example, disk packs, and it will be understood by those skilled in the art that the database system includes the required DASD controllers and channels that establish the communication between the database 8 and the processor 4. In another example the database (8) may be stored inside the main memory (4) itself.

Terminals 1 can be any Input/Output (I/O) device that accepts requests from and returns results to users. For example, the terminals 1 may be Personal Computers (PCs) or applications that use the advertised application programmer interface (API). Transactions and queries are submitted from the terminals 1 and are processed concurrently by the DBMS 6.

A transaction manager 5 is a software module of the DBMS 6 that controls how different transactions are conducted in the database 8. A Lock Manager (LM) 3 is a software

module of the DBMS 6 that helps the transactions in acquiring and releasing locks on data items in the database 8 in a controlled manner. The LM 3 makes sure that conflicting accesses to a given data item are not granted simultaneously. For example, if a simultaneous read and write access has been granted on a data item, then the reader of that data item may see an incomprehensible state of the data item. Thus some lock requests may not be granted to different transactions simultaneously, since there can be conflicts in the lock grant requests.

All transactions request that locks on data items be granted in a desired lock modes. For example, the reader may request a lock in shared (S) mode and a writer may request the same lock in an exclusive (X) mode. If S and X are defined not be compatible, then the LM 3 guarantees that no writer has access permission. In other words, no X lock exists on the data item if a S lock has already been obtained on that data item. Thus a reader can read a consistent state of the data item after S lock has been obtained on that data item.

Activity Duration Locking (ADL) module 9 allows the LM 3 to assign lock modes to groups of data associated with the same activities in a transaction. The ADL 9 uses an activity table 7 to keep track of the activities for different transactions.

Referring to FIG. 2, a transaction 12 may be viewed at different levels of abstractions. At the lowest level, transaction 12 is a set of reads and writes 15 on data items 16 which are stored in the database 8 (FIG. 1). At the intermediate level, the transaction 12 can be viewed as a collection of activities 14. Each activity 14 is a set of closely related actions on a group of data items 16. That is, an activity 14 is a set of reads and writes 15, and in turn is a subset of all actions performed by the transaction 12. For example, compilation of a structured query language (SQL) statement is an activity. Any transaction 12, as a whole, is itself an activity.

Each activity 14 spans a finite period in time. Activities 14 for transaction 12 may overlap in time, that is, they may be concurrent. Each activity 14 accesses a finite set of data items 16, and every data item 16 accessed by the transaction 12 belongs to a finite set of activities 14. The transaction 12 locks data items 16 through different activities 14. A transaction 12 can have arbitrary many concurrent activities 14 and any activity 14 may lock an arbitrary number of data items 16. These locks are released when the activity 14 finishes.

Referring to FIG. 3, the ADL module 9 provides an efficient means of classifying locks into activities with minimum overhead and also efficiently releases all locks in an activity, when the activity is completed. Classification and grouping of activities is performed in the lock manager 30. Thus, the transaction manager 5 only needs to note the currently alive activity identifiers for the activities.

For example, when transaction 12 starts a new activity 14 (activity X), the transaction 12 chooses a unique and unused token 34 for uniquely identifying the activity 14. This token 34 is referred to as the ActivityID. The LM 3 may help the transaction 12 in determining the unique ActivityID. When the transaction manager 5 requests the LM 3 for a lock, the ActivityID 34 is sent to the LM 3 so the LM 3 can classify the lock under the activity.

When the activity 14 is complete, the transaction 12 asks the LM 3 to release locks associated with the activity 14 identified by the unique ActivityID. The LM 3 releases all locks for the given activity 14 together. The ActivityID 34 is then declared free to be used by any new activities for transaction 12. Thus all locks associated with activity 14 are of this "activity duration".

Arbitrarily many lock durations can be defined and maintained in a non-serializable transaction 12. The onus of defining lock durations can be moved from the LM 3 to the transaction manager 5. This provides additional flexibility in defining and managing any number of desired lock durations. The ADL 9 provides a simple and efficient

implementation in which to define lock durations, enabling early releasing of locks for non serializable transactions.

FIG. 4 shows an example of how separating a transaction into activities helps efficiently complete a given task. Consider the notion of pre-fetching. A transaction 40 requests access to multiple tuples 42. These tuples 42 are retrieved from the database 8 (FIG. 1) one by one, or in bulk. For example, the tuples 42 may be retrieved in the context of a ODBC SQLFetch call made to an ODBC interface [ODBC SQL3.0].

In such an environment, the database application typically provides space for only one tuple into which the retrieved data can be copied. Thus the intermediate layer can retrieve each tuple one by one, copy the data into an application buffer, and return to the application.

Another choice is to retrieve the tuple IDs in bulk, and after the first fetch simply copy data from the current tupleID into the application buffer. This pre-fetching typically proves to be faster than retrieving each tuple one by one. For such a scheme, it becomes necessary to hold lock on each tuple in the pre-fetch group since modification to the pre-fetched tuples should be prohibited until data has been copied from the current tuple group. Again it is faster to unlock all tuples in a prefetch group in one operation rather than unlocking the tuples one by one. This need is ideally satisfied by the notion of activities.

Before prefetching the group of tuples 42, the transaction 40 starts an activity 44 and asks the lock manager 3 to associate the locks in the prefetch group 42A-42C with activity 44. After data from the last tuple 42C has been copied into the application buffers 46, the transaction 40 requests LM 3 to release all locks pertaining to the activity 44. This solution is much more efficient than maintaining a list of locks obtained in the given prefetch session in the transaction 40 and then finally unlocking these locks one by one.

FIG. 5 shows another example of how locks are associated with activities. The following transaction is conducted in the DBMS.

```
Transaction

    Execute (Activity #1)

        Data > 10 and Data < 16
        and ParentID = 4

        Execute (Activity #2)

            Set ParentID = ParentID +2,
            (confirm ParentID +2 exists, then change ParentID)

        End Activity #2
    End Activity #1
```

This transaction requests incrementing the ParentID for each data item having a ParentID of 4 and having a data value greater than 10 and less than 16. In one implementation of this transaction two steps are executed, in the first step all those tuples for which "data" value is between 10 and 16 are gathered, and in the second step this group is further qualified by selecting only those tuples that have ParentID = 4. The first step is implemented as an activity (Activity #1) that stores the tuples 62 for all data items having a value between 10 and 16 in a buffer 60. The tuples 62 contain an identifier (ID) that points to

associated data items. In this example, there may be ten thousand data items having values between 10 and 16. Under activity #1, a shared lock S is placed on the each one of the ten thousand tuples 62 satisfying Activity #1. A shared lock S means that certain other transactions, such as a read transaction, can access the same data items at the same time.

The next activity (Activity #2), under the same transaction, first finds all the tuples that have ParentID field set to 4. For all those tuples we then increment the ParentID field by 2. I.e. the second activity must first confirm that the ParentID is indeed 4, then Activity #2 increments the ParentID identified in the tuples by 2.

In this example, the tuples 64 identify the data items in buffer 60 that have a ParentID that can be incremented by two. An exclusive lock X is placed on the tuples 64 in buffer 60 identified under Activity #2. An exclusive lock X means that no other transactions can access the same data item at the same time. Thus, under activity #1 there are ten thousand S locks placed on the tuples 62 and under activity #2 there are only 500 X locks placed on the subset of tuples 64 (that had ParentID = 4).

After the end of Activity #2, there are 9500 data items that no longer need S locks. Activity #1 ends after the completion of Activity #2. The end of Activity #1 automatically releases the S locks on the 9,500 data items that are not verified under Activity #2.

FIG. 6 shows a sample lock 69 that includes a transaction ID 72, a lock count 74, a lock mode 76, other lock parameters 78 and an activity identifier 80. An activity bit map 82 is located in the transaction manager 5 or the lock manager 3 (FIG. 1) to track the different activities for a given transaction. The activity Id 80 contains a unique bit map value that identifies the activity holding the lock 69. A first activity #1 starts for a given transaction XID1. Since the current binary value in the activity map 82 is 00000, the activity #1 is assigned the binary value ActivityID = 00001. The first activity #1 may hold an S lock on associated data items. A second activity #2 is started and holds a X lock on associated data

items. The second activity #2 is assigned the next available bit in the activity bit map 82 and accordingly is assigned the ActivityID = 00010. The activity bit map 82 now has the binary value 00011. A third activity #3 holds an S lock on associated data items and is assigned the next available bit in the activity bit map 82 and accordingly is assigned the ActivityID = 00100. The activity bit map 82 then has the value 00111.

If the second activity #2 ends, the activity map 82 is changed to the binary value 00101. The next new activity to generate a lock 69 is then assigned the binary value ActivityID = 00010 made available by the termination of the second activity #2.

FIG. 7 describes in general terms how the ADL 9 in FIG. 1 operates. An activity is started by the transaction manager in block 70. The activity is identified according to the activity bit map in block 72. The data items referenced by the activity are locked according to the lock manager in block 74. The locks for the activities are tracked by the lock manager in block 76. As soon as an activity ends in block 78, the locks for the group of data items associated with that activity are released by the lock manager in block 80.

Thus it can be seen that the notion of activities is very valuable in the implementation of various optimizations and efficiently conforming to semantics of various isolation levels in DBMS. The current invention provides transaction managed durations that are very flexible.

The system described above can use dedicated processor systems, micro controllers, programmable logic devices, or microprocessors that perform some or all of the operations. Some of the operations described above may be implemented in software and other operations may be implemented in hardware.

For the sake of convenience, the operations are described as various interconnected functional blocks or distinct software modules. This is not necessary, however, and there may be cases where these functional blocks or modules are equivalently aggregated into a single logic device, program or operation with unclear boundaries. In any event, the

functional blocks and software modules or features of the flexible interface can be implemented by themselves, or in combination with other operations in either hardware or software.

Having described and illustrated the principles of the invention in a preferred embodiment thereof, it should be apparent that the invention may be modified in arrangement and detail without departing from such principles. I claim all modifications and variation coming within the spirit and scope of the following claims.